字符串基础

定义

字符集

一个 **字符集** Σ 是一个建立了全序关系的集合，也就是说，Σ 中的任意两个不同的元素 α 和 β 都可以比较大小，要么 $\alpha < \beta$，要么 $\beta < \alpha$。字符集 Σ 中的元素称为字符。

字符串

一个 **字符串** S 是将 n 个字符顺次排列形成的序列，n 称为 S 的长度，表示为 $|S|$。

如果字符串下标从 1 开始计算，S 的第 i 个字符表示为 $S[i]$；

如果字符串下标从 0 开始计算，S 的第 i 个字符表示为 $S[i-1]$。

子串

字符串 S 的 **子串** $S[i..j]$，$i \le j$，表示 S 串中从 i 到 j 这一段，也就是顺次排列 $S[i], S[i+1], \ldots, S[j]$ 形成的字符串。

有时也会用 $S[i..j]$，$i > j$ 来表示空串。

子序列

字符串 S 的 **子序列** 是从 S 中将若干元素提取出来并不改变相对位置形成的序列，即 $S[p_1], S[p_2], \ldots, S[p_k]$，$1 \le p_1 < p_2 < \cdots < p_k \le |S|$。

后缀

后缀 是指从某个位置 i 开始到整个串末尾结束的一个特殊子串。字符串 S 的从 i 开头的后缀表示为 $Suffix(S,i)$，也就是 $Suffix(S,i)=S[i..|S|-1]$。

真后缀 指除了 S 本身的 S 的后缀。

举例来说，字符串 `abcabcd` 的所有后缀为 `{d, cd, bcd, abcd, cabcd, bcabcd, abcabcd}`，而它的真后缀为 `{d, cd, bcd, abcd, cabcd, bcabcd}`。

前缀

前缀 是指从串首开始到某个位置 i 结束的一个特殊子串。字符串 S 的以 i 结尾的前缀表示为 $Prefix(S,i)$，也就是 $Prefix(S,i)=S[0..i]$。

真前缀 指除了 S 本身的 S 的前缀。

举例来说，字符串 `abcabcd` 的所有前缀为 `{a, ab, abc, abca, abcab, abcabc, abcabcd}`, 而它的真前缀为 `{a, ab, abc, abca, abcab, abcabc}`。

字典序

以第 i 个字符作为第 i 关键字进行大小比较，空字符小于字符集内任何字符（即：$a<aa$）。

回文串

回文串 是正着写和倒着写相同的字符串，即满足 $\forall 1\le i\le |s|, s[i]=s[|s|+1-i]$ 的 s。

字符串的存储

- 使用 `char` 数组存储，用空字符 `\0` 表示字符串的结尾（C 风格字符串）。
- 使用 C++ 标准库提供的 `string` 类。
- 字符串常量可以用字符串字面量（用双引号括起来的字符串）表示。

字符串匹配

字符串匹配问题

定义

又称模式匹配（pattern matching）。该问题可以概括为「给定字符串 S 和 T，在主串 S 中寻找子串 T」。字符 T 称为模式串 (pattern)。

类型

- 单串匹配：给定一个模式串和一个待匹配串，找出前者在后者中的所有位置。
- 多串匹配：给定多个模式串和一个待匹配串，找出这些模式串在后者中的所有位置。
  - 出现多个待匹配串时，将它们直接连起来便可作为一个待匹配串处理。
  - 可以直接当做单串匹配，但是效率不够高。
- 其他类型：例如匹配一个串的任意后缀，匹配多个串的任意后缀……

暴力做法

简称 BF (Brute Force) 算法。该算法的基本思想是从主串 S 的第一个字符开始和模式串 T 的第一个字符进行比较，若相等，则继续比较二者的后续字符；否则，模式串 T 回退到第一个字符，重新和主串 S 的第二个字符进行比较。如此往复，直到 S 或 T 中所有字符比较完毕。

```
// C++ Version
/*
 * s：待匹配的主串
 * t：模式串
 * n：主串的长度
 * m：模式串的长度
 */
std::vector<int> match(char *s, char *t, int n, int m) {
  std::vector<int> ans;
  int i, j;
  for (i = 0; i < n - m + 1; i++) {
    for (j = 0; j < m; j++) {
      if (s[i + j] != t[j]) break;
    }
    if (j == m) ans.push_back(i);
  }
  return ans;
}
```

时间复杂度

设 n 为主串的长度，m 为模式串的长度。默认 $m \ll n$。

在最好情况下，BF 算法匹配成功时，时间复杂度为 $O(n)$；匹配失败时，时间复杂度为 $O(m)$。

在最坏情况下，每趟不成功的匹配都发生在模式串的最后一个字符，BF 算法要执行 $m(n-m+1)$ 次比较，时间复杂度为 $O(nm)$。

如果模式串有至少两个不同的字符，则 BF 算法的平均时间复杂度为 $O(n)$。但是在 OI 题目中，给出的字符串一般都不是纯随机的。

字符串哈希(Hash)

定义:

我们定义一个把字符串映射到整数的函数， 这个 称为是 Hash 函数。我们希望这个函数 可以方便地帮我们判断两个字符串是否相等。

Hash 的思想:

Hash 的核心思想在于，将输入映射到一个值域较小、可以方便比较的范围。

这里的「值域较小」在不同情况下意义不同。

在 哈希表 中，值域需要小到能够接受线性的空间与时间复杂度。

在字符串哈希中，值域需要小到能够快速比较（10^9、10^{18} 都是可以快速比较的）。

同时，为了降低哈希冲突率，值域也不能太小。

性质

具体来说，哈希函数最重要的性质可以概括为下面两条：

1. 在 Hash 函数值不一样的时候，两个字符串一定不一样；
2. 在 Hash 函数值一样的时候，两个字符串不一定一样（但有大概率一样，且我们当然希望它们总是一样的）。

Hash 函数值一样时原字符串却不一样的现象我们成为哈希碰撞。

解释

我们需要关注的是什么？

时间复杂度和 Hash 的准确率。

通常我们采用的是多项式 Hash 的方法，对于一个长度为 l 的字符串 s 来说，我们可以这样定义多项式 Hash 函数：$f(s)=\sum_{i=1}^{l} s[i]\times b^{l-i} \pmod M$。例如，对于字符串 xyz，其哈希函数值为 xb^2+yb+z。

特别要说明的是，也有很多人使用的是另一种 Hash 函数的定义，即 $f(s)=\sum_{i=1}^{l} s[i]\times b^{i-1} \pmod M$，这种定义下，同样的字符串 xyz 的哈希值就变为了 $x+yb+zb^2$ 了。

显然，上面这两种哈希函数的定义函数都是可行的，但二者在之后会讲到的计算子串哈希值时所用的计算式是不同的，因此千万注意 **不要弄混了这两种不同的 Hash 方式**。

由于前者的 Hash 定义计算更简便、使用人数更多、且可以类比为一个 b 进制数来帮助理解，所以本文下面所将要讨论的都是使用 $f(s)=\sum_{i=1}^{l} s[i]\times b^{l-i} \pmod M$ 来定义的 Hash 函数。

下面讲一下如何选择 M 和计算哈希碰撞的概率。

这里 M 需要选择一个素数（至少要比最大的字符要大），b 可以任意选择。

如果我们用未知数 x 替代 b，那么 $f(s)$ 实际上是多项式环 $\mathbb{Z}_M[x]$ 上的一个多项式。考虑两个不同的字符串 s,t，有 $f(s)=f(t)$。我们记 $h(x)=f(s)-f(t)=\sum_{i=1}^{l}(s[i]-t[i])x^{l-i} \pmod M$，其中 $l=\max(|s|,|t|)$。可以发现 $h(x)$ 是一个 $l-1$ 阶的非零多项式。

如果 s 与 t 在 $x=b$ 的情况下哈希碰撞，则 b 是 $h(x)$ 的一个根。由于 $h(x)$ 在 $\mathbb{Z}_M$ 是一个域（等价于 M 是一个素数，这也是为什么 M 要选择素数的原因）的时候，最多有 $l-1$ 个根，如果我们保证 b 是从 $[0,M)$ 之间均匀随机选取的，那么 $f(s)$ 与 $f(t)$ 碰撞的概率可以估计为 $\frac{l-1}{M}$。简单验算一下，可以发现如果两个字符串长度都是 1 的时候，哈希碰撞的概率为 $\frac{1-1}{M}=0$，此时不可能发生碰撞。

```
// C++ Version
using std::string;

const int M = 1e9 + 7;
const int B = 233;

typedef long long ll;

int get_hash(const string& s) {
  int res = 0;
  for (int i = 0; i < s.size(); ++i) {
```

```
    res = (ll)(res * B + s[i]) % M;
  }
  return res;
}

bool cmp(const string& s, const string& t) {
  return get_hash(s) == get_hash(t);
}
```

Hash 的分析与改进

错误率

若进行 n 次比较，每次错误率 $\frac{1}{M}$，那么总错误率是 $1-\left(1-\frac{1}{M}\right)^n$。在随机数据下，若 $M=10^9+7$，$n=10^6$，错误率约为 $\frac{1}{1000}$，并不是能够完全忽略不计的。

所以，进行字符串哈希时，经常会对两个大质数分别取模，这样的话哈希函数的值域就能扩大到两者之积，错误率就非常小了。

多次询问子串哈希

单次计算一个字符串的哈希值复杂度是 $O(n)$，其中 n 为串长，与暴力匹配没有区别，如果需要多次询问一个字符串的子串的哈希值，每次重新计算效率非常低下。

一般采取的方法是对整个字符串先预处理出每个前缀的哈希值，将哈希值看成一个 b 进制的数对 M 取模的结果，这样的话每次就能快速求出子串的哈希了：

令 $f_i(s)$ 表示 $f(s[1..i])$，即原串长度为 i 的前缀的哈希值，那么按照定义有
$f_i(s)=s[1]\cdot b^{i-1}+s[2]\cdot b^{i-2}+\ldots+s[i-1]\cdot b+s[i]$

现在，我们想要用类似前缀和的方式快速求出 $f(s[l..r])$，按照定义有字符串 $s[l..r]$ 的哈希值为
$f(s[l..r])=s[l]\cdot b^{r-l}+s[l+1]\cdot b^{r-l-1}+\ldots+s[r-1]\cdot b+s[r]$

对比观察上述两个式子，我们发现 $f(s[l..r])=f_r(s)-f_{l-1}(s)\times b^{r-l+1}$ 成立（可以手动代入验证一下），因此我们用这个式子就可以快速得到子串的哈希值。其中 b^{r-l+1} 可以 $O(n)$ 的预处理出来然后 $O(1)$ 的回答每次询问（当然也可以快速幂 $O(\log n)$ 的回答每次询问）。

练习:

1. 洛谷U248425允许k次失配的字符串匹配

2. 洛谷U248467最长回文子串

3. 洛谷U248544最长公共子字符串